# On the implementation of MCMC proposals over Stochastic Logic Programs

Nicos Angelopoulos James Cussens
Department of Computer Science, University of York, UK
{nicos,jc}@cs.york.ac.uk

**Abstract**

We present some implementation issues that arise in an MCMC algorithm which uses a probabilistic extension of Logic Programming. In particular, we are concerned with the challenges presented in extending the underlying logic engine to facilitate such algorithms. We identify the core issues and analyse the possibility of integration with Prolog. Wherever possible we try to address the generic probabilistic behaviour of each issue which may be useful to other statistical tasks in addition to the central MCMC algorithm.

## 1 Introduction

A stochastic logic program (SLP) is a labelled logic program which, in combination with a goal, defines a log-linear distribution over refutations, i.e. successful derivations, of that goal. Consequently, it defines a marginal distribution over instantiations of the variables in the goal. SLPs generalise hidden Markov models (HMMs) and stochastic context-free grammars (SCFGs) by replacing parsing with proof search and unification. In the context of probabilistic models, SLPs belong to the 'generative' approach often closely related to grammars. A number of other formalisms in the same category also elaborate some underlying logical paradigm with probabilistic concepts, for instance [8, 9].

Integrating probabilistic and logical reasoning is an important area of research. The probabilistic part, however, covers a broad spectrum of tasks such as probabilistic, statistical and stochastic reasoning. It seems that there is no universal methodology that will be able to efficiently compute all interesting statistical questions. The kind of integration seems to be highly dependent on the statistical task at hand. Statistical questions, or tasks, which are important for integrative approaches include: finding the probability of statements, sampling (i.e. probabilistically finding an instance in the probability space), parameter estimation (e.g. [3] [9]), stochastic searching (e.g. simulated annealing) and Bayesian learning algorithms.

MCMC algorithms are widely used Bayesian learning algorithms. Such an algorithm operating over SLPs is analysed in [2]. This version of the MCMC performs transitions between leaves of a stochastic SLD-tree. The operations needed to do so are hard to implement in the standard logic engine. In this paper, we identify the main operations involved and explore how the logic engine might be modified to incorporate these. Wherever possible we try to address the generic probabilistic behaviour of each issue which may be useful to other statistical tasks in addition to the MCMC algorithm.

The remainder of the paper is structured as follows. Section 2 and Section 3 present Extended SLPs and an MCMC algorithm which operates over stochastic programs, respectively. Section 4

```
1/2 :: nat( 0 ).
1/2 :: nat( s(X) ) :- nat( X ).
```

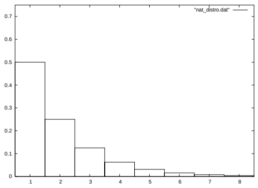

Figure 1: SLP and associated distribution over generated terms.

discusses clause selection under SLPs. Section 5 relates recursion to aspects of statistical behaviour. Section 6 presents the centre-piece of this paper; a MCMC backtracking scheme. Section 7 explores the exploitation of independence for model generating programs. Section 8 discusses tabulation in relation to the MCMC algorithm. The concluding remarks are in Section 9.

## 2 Extended Stochastic Logic Programs

Stochastic Logic Programs (SLPs) extend Logic Programming by allowing probabilistic labels to be attached to clauses. In this paper we will broadly follow the semantics presented in [4] which define probabilities over computed answers. However, here we are concerned with describing the general computational concepts involved rather than examining a particular kind of semantics. For example consider the simple SLP program in Fig. 1 defining a geometric distribution over natural numbers. Each instantiation of $X$ in query $nat(X)$ against this program represents a natural number. A probability can be attached to each instantiation. The program defines $P(X = 0) = 1/2, P(X = 1) = 1/4$ and in general, $P(X = i) = 2^{-i-1}$.

More formally, let $l_i$ denote the label of the $i$th clause ($C_i$) and let $\lambda$ be the vector of logs of the labels: $\lambda = (\lambda_1, \dots, \lambda_n) = (\log l_1, \dots, \log l_n)$. Given a goal $G$, consider $x$ a derivation which begins with $G$. $\psi_\lambda(x)$ is the product of the labels $(l_{x,1}, \dots l_{x,k})$ for the clauses $(C_{x,1}, \dots C_{x,k})$ used in the derivation. This is simply the statistically intuitive product of sampling from independent choices. Thus,

$$\psi_\lambda(x) = \prod_{i=1}^{k} l_{x,i} = \exp \sum_{i=1}^{k} \lambda_{x,i}$$

Based on $\psi_\lambda$(x) distributions over refutations and yields (instantiations of $G$) can be defined as described in [3]. There, only labels that are constant numbers were considered. In contrast, here we consider an extention to the language which allows arbitrary guarded calls to feed into expressions which are evaluated when a goal is matched to a clause. We introduce this syntax by means of an example:

```
:- pvars( umember(_El,List), [L-(length( List, L ),L>0)] ).

1/L :: [L] :: umember( El, [El|T] ).
1 - (1/L) :: [L] :: umember( El, [_|T] ) :-
                [L-1] :: umember( El, T ).
```

The intuition behind `umember/2`, is that the probability with which an element is picked from the list is equal to 1 divided by the length of the list. Thus at call time, $length(List, L), L > 0$ will be used to compute $L$, this in turn will be used to compute the labels of the clauses via the expressions $1/L$ and $1-(1/L)$. The first argument for directive `pvars/2` is the head corresponding to a probabilistic predicate while the second one is a list of pairs. Each member in the list pairs a variable to its guard. The syntax for defining probabilistic clauses is extended to, `Expr::List::Clause`. *Expr* is an arithmetic expression containing some of the variables from *List*. *List* is a list of variables, each of which corresponds to one item in the second argument of the relevant `pvars/2` directive. `Clause` is a standard clause. Note that the recursive call in our example is '`[L - 1]::umember( El, T )`'. Since the length of tail $T$ can be computed from that of $[H|T]$, recursive calls can avoid calling `length/2`. In the example, this is achieved by attaching the evaluable expression 'L - 1' to the recursive call. In general a goal corresponding to a probabilistic call can be either of the form *Head* or of the form *Exprs* :: *Head*. With *Exprs* being a list of evaluable expressions and variables. When the call is of the form *Head* or when a variable is found in *Exprs*, the relevant guards from `pvars/2` are called. Otherwise, the expressions in *Exprs* are evaluated.

The declarative reading of `umember/2` is identical to standard `member/2`. However, it also has a statistical reading which says that each member of the list is picked with equal probability. Provided the expressions for each clause of a predicate involved in a refutation step, evaluate to numbers $l_1, \ldots, l_n$ such that $0 \leq l_i$ and $\sum_i l_i = 1$, then it is easy to show that semantics of SLPs also apply to the extended version.

# 3 MCMC

SLPs can be used to generate all possible (statistical) models in a model space (e.g. space of all Bayesian Networks). The probabilistic labels define the prior distribution over possible models. The benefit of using a logic based formalism is that the theories can be expressed at a high level of abstraction. The logical part can be used to define the space of possible models. The probabilistic part allows us to express preferences and prior expectations about the probability of each model. For example consider a logic program defining Bayesian Networks (BNs). This program constructs all admissible BNs which are possible for a given number of nodes. Posing the query $bn([1,2,3], Bn)$ to the program will instantiate $Bn$ to a term structure representing some three node BN. Each BN has an associated probability (of derivation, or sampling) which is its sampling probability according to the program. Examples of SLP programs defining BN theories can be found in [2].

Markov chain Monte Carlo (MCMC) algorithms [6] are commonly used to estimate $P(M \mid D)$ (posterior probability of models given data $D$) from a prior distribution $p(M)$. Provided some weak conditions are met, and that the Markov chain is run for a sufficiently long period, then the visiting frequencies approach the true posterior probability of models. The posterior probability of a model is the probability that the particular model is the 'true' model. For an introduction of MCMC for machine learning see [1]. In the case of SLPs p(M) is equal to the sampling distribution. In constructing the Markov chain we assume the Metropolis-Hasting algorithm

0. Set $i = 0$ and find $M_0$, an initial leaf of the SLD-tree, using the prior.
1. From $M_i$ produce a candidate leaf $M_*$. Let the probability of reaching $M_*$ be $q(M_i, M_*)$.
2. Let
$$\alpha(M_i, M_*) = \min\left\{\frac{q(M_*, M_i)P(D|M_*)P(M_*)}{q(M_i, M_*)P(D|M_i)P(M_i)}, 1\right\}$$
$$M_{i+1} = \begin{cases} M_* & \text{with probability } \alpha(M_i, M_*) \\ M_i & \text{with probability } 1 - \alpha(M_i, M_*) \end{cases}$$
3. If $i$ reached limit then terminate, else set $i = i + 1$ and repeat from 1.

$q(M_i, M_*)$ is the transition probability, i.e. the probability of proposing $M_*$ from $M_i$, $P(D \mid M)$ is the likelihood that the data were generated by $M$ and $P(M)$ is the prior probability of $M$.

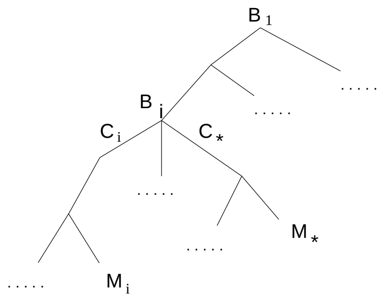


Figure 2: Reaching $M_*$ from $M_i$ in the SLD-tree.

Since SLPs can be used to generate all possible models, we can impose the MCMC transitions on the SLD-tree. However, here, the SLD-tree is not explored depth first, left to right. It is also the case that probabilities label some of the edges of the tree.

The essence of this approach is that at iteration $i$ we propose $M_*$ from $M_i$ by means of operations on the SLD-tree. First we backtrack to $B_i$ , an ancestor node of $M_i$ , and then sample from that node onwards. A general case of this is illustrated in Fig. 2. The exact way $M_*$ is chosen can vary and it will affect the convergence properties for the Markov chain. It is thus important that any implementation must provide flexibility in choosing different proposal backtracking strategies.

Example algorithms for implementing $q(M_i, M_*)$ include: (a) backtrack one point at the time, with fixed probability $p_b$, (b) choose $B_i$ uniformly from the available points, (c) adopt a strategy which favours sections of the path, such as nearer the root or nearer the current leaf, or (d) always backtrack to $B_1$, which is known as the independent sampler.

# 4 Clause selection

Execution in Prolog proceeds by matching a goal to clauses according to their textual order in the source file. The presence of probability labels in SLPs, though, make it possible to select clauses depending on their labels. This is an operation necessary in various statistical tasks.

In general a probabilistic predicate defined by $n$ clauses will have labels $l_1, \ldots, l_n$ with $\sum l_i = 1$. In the case where some $l_i$ is an expression, the first step will have to be its evaluation to a number. Selecting one of the clauses ($C_i$) according to the $l_i$s can the be reduced to selecting a random number, $RN_1 \in (0, 1]$ and picking $l_i$ such that $\sum_{j<i} l_j < RN_1$ and $RN_1 \leq \sum_{k \leq i} l_k$.

If alternatives for $C_i$ need to to be considered, i.e. due to some form of backtracking, the process is repeated, with $l_i$ removed and a new random number selected. The process continues for each backtracking until no further clauses are left. For each revisited time let the set of remaining labels be $L$. We then need to revise the procedure for selecting a label based on a random number, $RN_2$ since $\sum_{l_j \in L} l_j \leq 1$. $RN_2$ should be in $(0, \sum_{l_j \in L} l_j]$. This is a form of renormalising the attached probabilities. In effect we let updated $l'_i = \frac{l_i}{\sum_{l_j \in L} l_j}$. This is a common technique in statistical reasoning.

## 4.1 Failure

In terms of logic, a path that leads to a failure is uninteresting and ignored, with the computation continuing from the earliest backtrack point. For some statistical tasks, such as parameter estimation, the failed path is an important part of the proposed algorithms, and can contribute to the estimated labels. In such tasks, the probability of the path, flagged by the fact that it corresponds to a failed computation is useful information which needs to be passed to the user.

The MCMC scenario we have presented, can work with standard backtracking and with explicit representations of failure. By explicit failure we mean that when a failure is encountered during derivation, i.e. non empty goal with non matching predicate to perform a reduction step, the engine should, instead of backtracking, succeed while setting a special flag to 'fail'. This kind of failure also simplifies the mathematics of the MCMC. However, failed computations are pointless in terms of model construction, thus the Markov chain constructed will have slower convergence (each failed computation corresponds to a do-not-jump cycle).

For example consider sampling $N$ samples of the goal $p(X)$ against program:

```
1/2 :: p( a ).                        1/3 :: q( b ).
1/2 :: p( X ) :- q(X), t(X).          2/3 :: q( c ).          1 :: t(b).
```

According to explicit failure we will observe the following sampling distribution $\{a - 1/2, b - 1/3, fail - 1/6\}$. In contrast, when using backtracking, the sampling distribution is $\{a - 1/2, b - 1/2\}$. With backtracking, the mathematical analysis of MCMC becomes harder, since one has to be precise about the shift of the probability density which is due to the backtracking. In our example there is a shift of 1/6 towards $p(b)$. On the other hand, backtracking is much more efficient for MCMC experiments since there is no wasted cycles due to hitting a failure. This is an important issue since many model spaces can be intuitively encoded as generate and test programs which include a significant proportion of failures.

# 5 Recursion

## 5.1 Amplification

Assume we are working with a recursive SLP predicate defined by the two clauses:

```
l1 :: name1( ... ) :-                 l2 : name1( .... ) :-
        ... .                              ... ,
                                           name1( ... ),
                                           ... .
```

When using a random number ($RN_1$) for choosing between $l1$ and $l2$ we can further *amplify* this number to determine all future calls to `name1` . In particular, the recursive call amplifies $RN_1$ by multiplying it with $1 - l2$ to produce $RN_2$. The second time `name1` is entered the first clause is chosen, if $RN_2$ is less than $l1$, otherwise the second clause is chosen, in which case, the process is repeated.

## 5.2 Tail recursion

Let a probabilistic choice point be a choice point involving a probabilistic predicate. For tail recursive predicate definitions, such as

```
l3 : name2( ... ) :-
      ... .
l4 : name2( .... ) :-
      ...,
      name2( ... ).
```

it might be possible to use amplification to reduce the number of probabilistic choice points. This is akin but not the same as tail recursion optimisation in Prolog, where no choice points are left behind. Here, we explore the idea of reducing the N backtrack points, left behind by the recursive calls to a single choice point. This optimisation improves the statistical behaviour of the program, and might improve execution performance.

A task in which amplified tail recursion might be useful is in MCMC model explorations. Having a single backtrack point for recursive predicates can lead to more compact path representations, leading to more precise proposal backtracking to an arbitrary point in the path.

Note, that if in addition, the proposal backtracking used for an MCMC does not require exclusion of the visited branch (chosen label of the current model for the point where proposal backtracking stops travelling upwards, e.g. see [2]) then we have a further departure from traditional LP way of thinking about backtracking. The algorithm allows 'backtracking' to the same model, of the same leaf on the SLD tree.

### 5.3 umember/2

Recall the program:

```
:- pvars( umember(_El,List), [L-(length( List, L ),L>0)] ).

1/L :: [L] :: umember( El, [El|T] ).
1 - (1/L) :: [L] :: umember( El, [_|T] ) :-
                [L - 1] :: umember( El, T ).
```

The non-optimised stochastic SLD (SSLD) tree for query '`?- umember( X, [a,b,c]).`' against the above program might be visualised as shown in Fig. 3

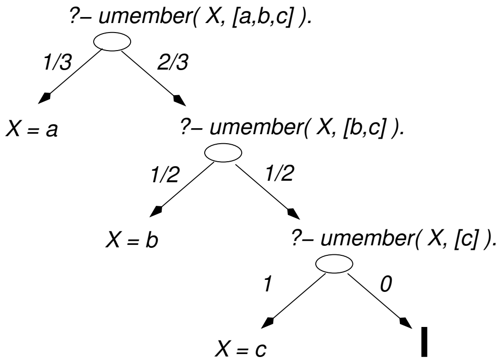


Figure 3: Non-optimised SSLD

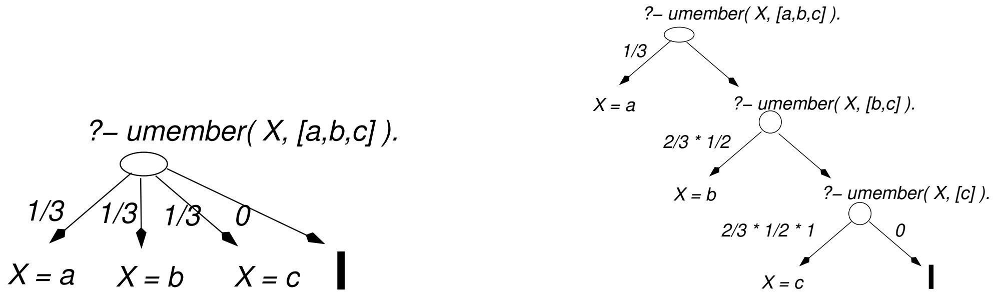


Figure 4: Optimised SSLD. Left: its statistical view. Right: its procedural view

The statistical and procedural views of the optimised SSLD for the same query are shown in Fig. 4. In the first case, the optimisation manages to present the operation of selecting a member of the list uniformly, as an atomic, statistical operation. This is an important issue in MCMC since the number of backtracking points affect the compactness of the search space. If a predicate representing an atomic operation leaves a number of backtrack points then it becomes more likely that one of these will be selected by the backtracking strategy, thus altering the selection likelihood.

In the procedural view, we show how the optimisation actually works. Circles correspond to predetermined choices and the probabilities on the branches are expressions that divide a single probability space. There is a single backtrack point which corresponds to selecting a single random number. Following which, the selection of $X$ can be done in a non-probabilistic fashion.

# 6 Backtracking and model comparison

Prolog's standard backtracking strategy is ill equipped to deal with MCMC proposals. The standard strategy explores the SLD tree depth-first, returning one answer at a time. Instead, in MCMC proposals, backtracking is to an arbitrary point in the path, and comparison of two leaf nodes is required. The latter means that two replies to a query should be concurrently present in memory.

A path is the unique ordered list of probabilistic choice points that lead from the root of the SLD to a leaf node. In Fig. 2 two paths are shown in some detail, those leading to $M_i$ and $M_*$ . Let $B_1, \ldots, B_i, \ldots, B_{li}$ be the backtrack points for $M_i$ and $B_1, \ldots, B_i, B^*_{i+1} \ldots, B^*_{l*}$ the backtrack points for $M_*$ . Then an algorithm for managing the low level operations required by the MCMC at the LP engine level is:

10. sample initial goal deriving $M_1$.

20. backtrack to arbitrary $B_i$ and sample $M_*$

    - do not destroy choice points $B_{i+1} \ldots B_{l_i}$
    - add $B^*_{i+1} \ldots B^*_{l_*}$ as $B_{l_i+1} \ldots B_{l_i+l_*-i}$

30. set $M_{i+1}$ to either $M_i$ or $M_*$ ; reclaim memory for

    - $B_{i+1} \ldots B_{l_i}$, if $M_{i+1} = M_i$
    - or $B_{l_i+1} \ldots B_{l_i+l_*-i}$, if $M_{i+1} = M_*$

    unless termination conditions have been reached, go to 20

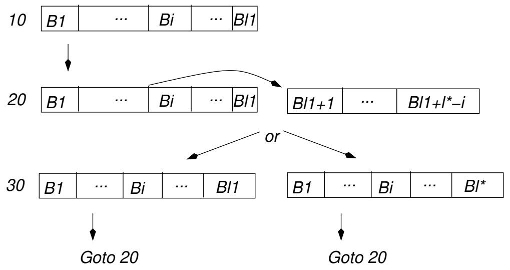


Figure 5: Memory allocation scheme

Fig. 5 illustrates this in terms of a simplistic memory allocation scheme. A user level interface could look like:

- sample( +Goal, +DefStrategy, -Instance, -GHandle, -IHandle, )
- backtrack( +GHandle [,+Strategy], -Instance, -IHandle )
- discard( +Handle )

The arguments are: *Goal* for a top level goal such as $bn([1,2,3], Bn)$. *DefStrategy* for the default backtracking strategy. *Instance* is a computed answer for *Goal*, such as $bn([1,2,3],[1-[2],2-[3],3-[]])$. *GHandle* is a goal handle, which should be a unique system identifier. *IHandle* is the instance handle, which should be an identifier to memory allocations for $B_{i+1}\dots B_{l_i}$ and $B_{l_i+1}\dots B_{l_i+l_*-i}$ for $M_i$ and $M_*$ respectively. *Strategy* is a strategy to be used for a particular backtrack operation. It can override *DefStrategy* to obtain a new leaf instantiation with `backtrack/4`. *Handle* can be either a goal handle or an instance handle. The presented MCMC algorithm can then be written in Prolog, as:

```
mcmc(Goal, Stg, Visited) :-
   sample(Goal, Stg, M1, GHandle, M1Handle),
   mcmc_iterations(true, GHandle, M1, M1Handle, Visited),
   discard( GHandle ).

mcmc_iterations(false, _GHandle, M, IHandle, [M]) :-
   discard(IHandle),
mcmc_iterations(true, GHandle, Mi, MiHandle, [Mi|TVisited]) :-
   backtrack(GHandle, Mstar, MsHandle),
   jump(Mi, MiHandle, Mstar, MsHandle, Mst, MstHandle, DHandle),
   discard(DHandle),
   termination(Cond),
   mcmc_iterations(Cond, GHandle, Mst, MstHandle, TVisited).
```

Note, that user defined backtracking strategies are possible. As long as a simple representation of the path is maintained, the selection of a particular backtrack point, can be dictated by user-defined predicates.

# 7 Independent branches

When considering proposal distributions over LP with probability labels, we view the SSLD as the structure on which movements are made. A model corresponds to a leaf of such trees, and proposing a new model entails the undoing of all choices up to the least distant ancestor.

```
1 - 1/D :: [D] :: cart(leaf).
1/D :: [D] :: cart(Split-[L,R]) :-
        NewD is D+1,
        choose_split(Split),
        NewD :: cart(L),
        NewD :: cart(R).

choose_split(x1).
choose_split(x2).
choose_split(x3).
```

Figure 6: Simple C&RT generator

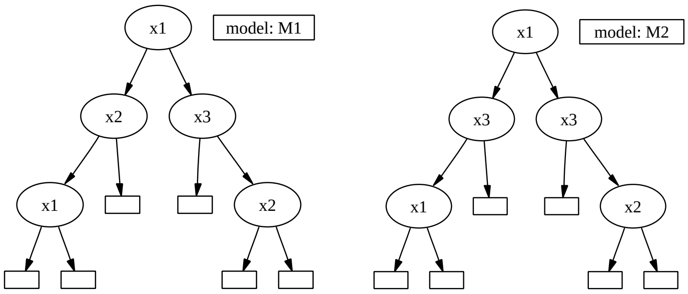


Figure 7: C&RT models, $M_1$ and $M_2$

Consider the program in Fig. 6. This program can be used to generate Classification and Regression trees (C&RT) such as models $M_1$ and $M_2$ of Fig. 7. Although the two models are fairly 'similar', their disagreement being a single node, they are quite far in terms of positions on the SLD tree. As a result $q(M_1, M_2)$ for any proposal backtracking strategy which depends on the location of $M_1$ will always be disproportionately, or at least counter-intuitively, small.

The part of the SSLD that generates the subtrees starting to the right child of the top node, for $M_1$ and $M_2$, is shown in the left hand side of Fig. 8. The subtree is identical in both models and is labelled by `x3`. Part of the (infinite) SSLD tree generating all possible C&RTs according to the program in Fig. 6 is shown in Fig. 9. The paths that lead to $M_1$ and $M_2$ are shown at length. The common subtree of Fig. 8 starts at the two nodes labelled by 'lft_4'. To reach $M_2$ from $M_1$, and vice versa, a leaf dependent proposal backtracking strategy would have to travel upwards until reaching node 'choose' at depth 4. Furthermore, forward sampling must be employed to re-built the shared branch before reaching model $M_2$. This last step will also impede independent samplers.

It is thus clear that although $M_1$ and $M_2$ are intuitively speaking, neighbours, they do not belong to the same tree neighbourhood. In what follows we will try to propose a remedy to this problem. However, it is worth noting that it is extremely hard to justify and quantify the effect of different neighbourhood groupings on the efficiency of MCMC. Any comparative studies are likely to be experimental in nature.

A path in the SSLD tree can be viewed, in an alternative manner, as a proof tree ([7], p53). The proof trees for $M_1$ and $M_2$ are given in the right hand side of Fig. 8. Backtracking can then be seen as selecting internal nodes in these trees and providing alternative components to the proof.

The key observation is that branches at the same depth of the proof tree are independent if they do not share uninstantiated variables. It would thus be desirable to be able to determine that the subtrees starting at `x3` and `x2` in $M_1$ are independent and changes to one of them do not need to involve changes to the other.

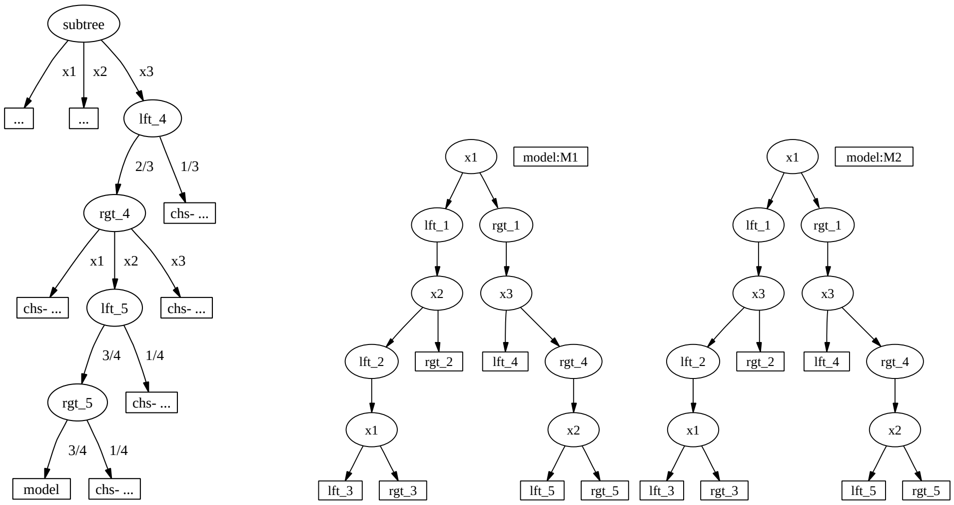


Figure 8: Left: Common branch of $M_1$ and $M_2$ . Right: Proof trees for $M_1$ and $M_2$ .

In general when considering recursive structures as statistical models generated from a probabilistic LP engine, it might beneficial to identify independent branching in the programs that describe the constructive theories. This may lead to formations of more intuitive neighbourhoods, and backtracking that can operate on parts of the structure depending on their independence relation rather than the sequential order in a path. Finally it might lead to more flexible algorithms that perform additions and deletions over entire branches. In the case of C&RT models such operations have been suggested as possible improvements on current algorithms, for example see [5]. However, there is no current technology to achieve this.

# 8 Tabling and control

Strictly non-probabilistic predicates are non-probabilistic predicates which contain no direct, or indirect, calls, in their bodies, to a probabilistic predicate. In addition, they should be free of side-effects. For most statistical tasks a number of restrictions on the success set of such predicates is placed to insure sound semantics. The most restrictive condition as in [4] is that each such predicate should succeed at most once for every path uniquely identified by the ordered list of clauses for probabilistic clauses used in the path.

Irrespective of the particular proposal mechanism, performance improvement may be possible from tabling calls to strictly non-probabilistic predicates. Tabling will allow sharing across branches (as in the case of Prolog) but also would improve performance on revisited branches. Recall that MCMC does not perform a depth first exhaustive search, thus, it is possible, indeed highly probable, that visited leaves share a common path prefix. If the common prefix includes calls to strictly non-probabilistic clauses, then normal Prolog style tabling [10] of these calls, will improve performance.

If the proposal backtracking is implemented at a meta-level, strictly non-probabilistic calls can be indexed away and reused. Revisiting Fig. 2, let $T_1, \dots T_i$ be the strictly non-probabilistic calls in path to $M_i$ $T_1, \dots T_*$ the ones in the path to $M_*$ and $T_1, \dots T_{B_i}$ the common calls. In a non-native implementation of the MCMC proposal getting to $M_*$ from $M_i$ might well involve guiding the top goal from $B_1$ to $B_i$ and then sampling freely according to the prior. Having stored $T_1, \dots T_{B_i}$

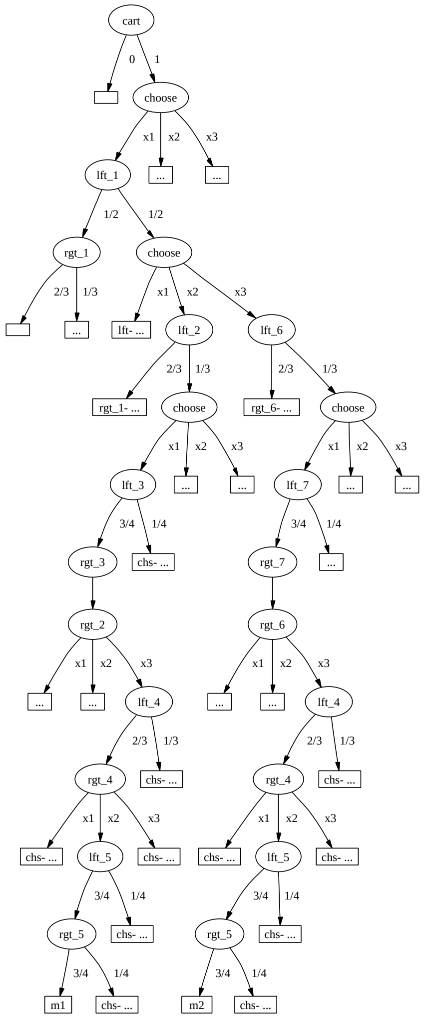


Figure 9: Generator SSLD for $M_1$ and $M_2$

can make the guiding processes much more efficient than recomputing everything. Indexed calls $T_1, \ldots T_*$ can be similarly valuable at iteration $i+1$ if $M_{i+1} = M_*$.

Part of the proposal scheme may be simulated by runtime predicates. Backtracking to an arbitrary point can be achieved via attaching a unique identifier to each probabilistic choice point, $B_j$. The desired identifier is posted just before explicit failure is issued. Each backtrack point reached, before the assigned one, inspects the posted identifier and fails. Note that this, (a) does not allow two paths to be present at the same time, i.e. if $M_{i+1} = M_i$ the engine needs to re-fail and use stored information to guide computation at the same position, and (b) this will be an improvement to re-starting from top, only in the cases where shallow backtracking is required, i.e. $B_i$ is nearer to $B_{l_1}$ than to $B_1$.

## 9 Conclusions

Adding some probabilistic element to logic programming has been explored in a variety of ways since the inception of logic programming. One of the emerging characteristics is that different formalisms might be appropriate depending on the statistical tasks for which an extended language is to be used. In this paper we have analysed the behaviour of one formalism in one statistical task. The formalism provides a rich language for expressing probabilistic distributions over theories of statistical models, while the MCMC algorithm provides a versatile tool for Bayesian learning.

Our objective has been to describe the ways in which a Prolog engine can be improved for the statistical task at hand. Our main aim has been to analyse the various issues involved in a manner that is accessible to researchers that are not familiar with MCMC or similar statistical tasks. We have also included particular suggestions on how the logic engine can provide better support for the MCMC algorithm. To the extent possible, we have tried to treat each issue in the most general way, in relation to other statistical tasks.

## Acknowledgements

Thanks to Taisuke Sato and Yoshitaka Kameya for useful discussions. This work was supported by EPSRC grant GR/S30993/01 *Stochastic Logic Programs for MCMC*.

## References

[1] C. Andrieu, N. de Freitas, A. Doucet, and M. I. Jordan. An introduction to mcmc for machine learning. *Machine Learning*, 50(1-2), January-February 2003.

[2] N. Angelopoulos and J. Cussens. Markov chain Monte Carlo using tree-based priors on model structure. In *Seventeenth UAI Conference*, pages 16–23, 2001.

[3] J. Cussens. Stochastic logic programs: Sampling, inference and applications. In *Sixteenth UAI Conference*, pages 115–122, San Francisco, USA, 2000.

[4] J. Cussens. Parameter estimation in stochastic logic programs. *Machine Learning*, 44(3):245–271, 2001.

[5] H. A. Denison, B. K. Mallick, and A. F. M. Smith. A Bayesian CART algorithm. *Biometrika*, 85:363–377, 1998.

[6] W. R. Gilks, S. Richardson, and D. J. Spiegelhlater, editors. *Markov Chain Monte Carlo in Practise*. Chapman & Hall, 1996.

[7] U. Nilsson and J. Maluszynski. *Logic, Programming and Prolog*. Second edition, 1995.

[8] S. Riezler. *Probabilistic Constraint Logic Programming*. PhD thesis, Universität Tübingen, 1998. AIMS Report 5(1), 1999, IMS, Universität Stuttgart.

[9] T. Sato and Y. Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.

[10] D. S. Warren. Memoing for logic programs. *Comm. of the ACM*, 35(3):94–111, March 1992.